Kubernetes, commonly known as K8s, is an open-source container orchestration platform that simplifies the deployment, scaling, and management of containerized applications. It provides a robust and scalable framework for automating the management of containerized workloads and services. To fully grasp the capabilities of Kubernetes, one must understand its key components. In this article, we will explore the essential components of Kubernetes in detail and understand their roles in creating a highly available and resilient application infrastructure.

**1. Master Components:**

The master components are responsible for managing the overall Kubernetes cluster. They include:

a) API Server: The API server acts as the central control point, serving as the gateway for all administrative and management operations. It exposes the Kubernetes API and handles API requests, authentication, and authorization.

b) Scheduler: The scheduler assigns workloads to the appropriate nodes based on resource availability, constraints, and policies. It takes into account factors like CPU and memory requirements, node affinity, pod anti-affinity, and workload priorities.

c) Controller Manager: The controller manager ensures that the desired state of the cluster is maintained by constantly monitoring and reconciling the actual state with the desired state. It includes various controllers, such as the Replication Controller, Deployment Controller, and StatefulSet Controller, to manage and maintain the desired state of pods and other resources.

d) etcd: etcd is a distributed and highly available key-value store that stores the cluster's configuration data, state, and metadata. It provides a reliable and consistent data store for the entire cluster, allowing components to store and retrieve information about the cluster's configuration and state.

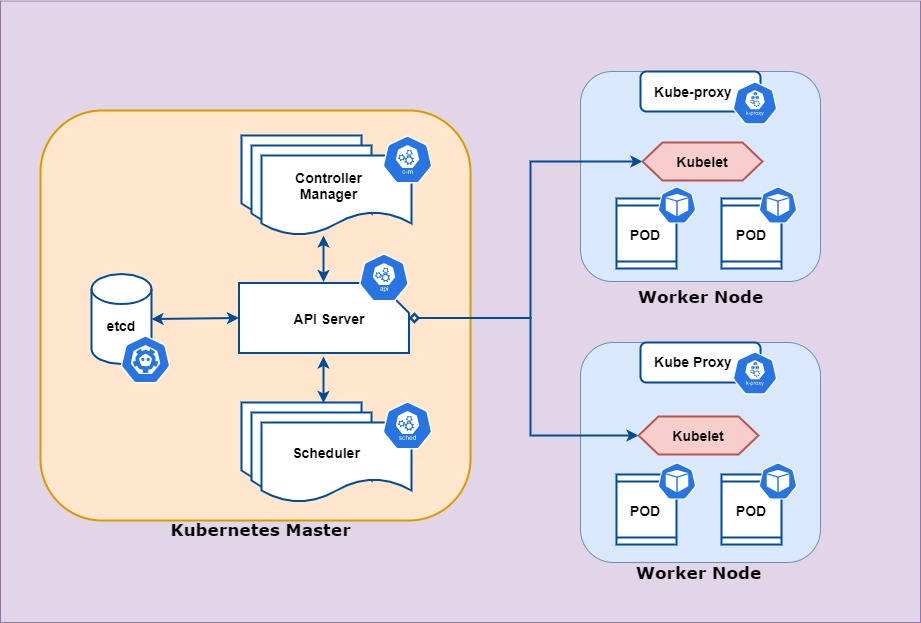
**2. Node Components:**

The node components run on every worker node and are responsible for executing and managing the application workloads. They include:

a) Kubelet: The kubelet is an agent that runs on each node and is responsible for managing containers, ensuring they are running and healthy. It communicates with the API server to receive instructions and updates about the desired state of pods and takes necessary actions to maintain that state.

b) kube-proxy: kube-proxy is responsible for network proxying and load balancing. It routes traffic to the appropriate containers based on IP addresses and ports. It also handles service discovery within the cluster, allowing pods to communicate with each other using service names.

c) Container Runtime: The container runtime, such as Docker or containerd, is responsible for pulling and running container images. It provides an environment where containers can run securely and efficiently.



**3. Add-Ons:**

Add-ons are optional components that enhance the functionality of Kubernetes. Some commonly used add-ons are:

a) DNS: The DNS add-on provides a DNS service for internal service discovery within the cluster. It assigns DNS names to services and allows pods to communicate with each other using these names.

b) Dashboard: The Kubernetes dashboard provides a web-based user interface for managing and monitoring the cluster. It allows users to view and manage resources, troubleshoot issues, and monitor the health and performance of the cluster.

c) Ingress Controller: An Ingress Controller allows inbound traffic to reach services within the cluster by implementing rules and load balancing. It acts as a reverse proxy and handles routing of external requests to the appropriate services.

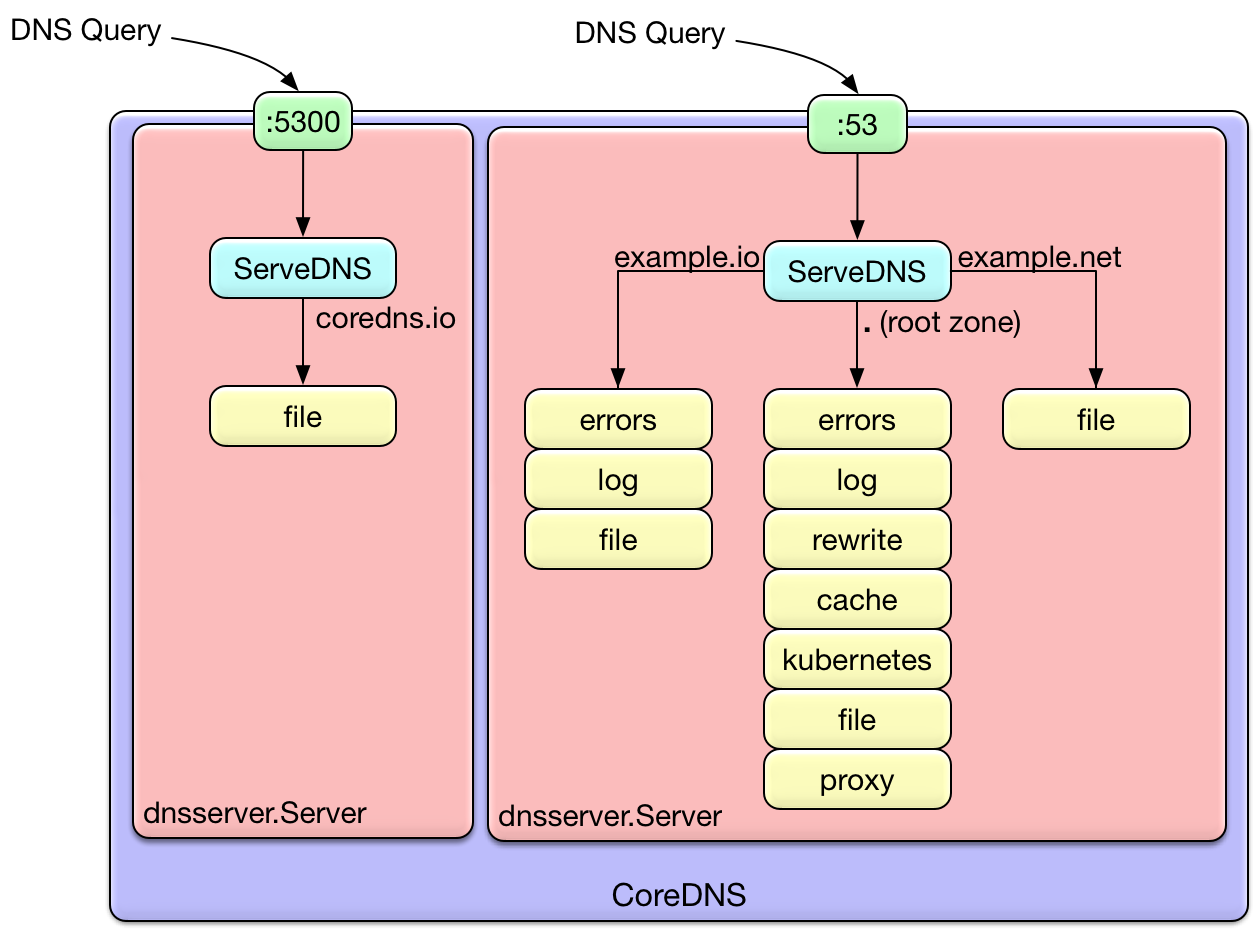
d) Storage Plugins: Storage plugins enable persistent storage for stateful applications running on Kubernetes. They allow pods to request and use persistent volumes, which provide durable storage that is independent of the pod lifecycle.

4. Persistent Volumes:

Persistent Volumes (PVs) are a way to provide durable storage for containers. They are independent of the pod lifecycle and can be dynamically provisioned or statically defined. PVs allow data to be stored persistently, even if the pod is restarted or rescheduled to a different node.

CoreDNS

In Kubernetes, reliable and efficient service discovery is crucial for seamless communication between pods and services. CoreDNS, a flexible, extensible, and lightweight DNS server, has become the default DNS solution in Kubernetes. It simplifies the process of service discovery within the cluster, enabling pods and services to communicate with each other using user-friendly names. In this article, we will explore CoreDNS in Kubernetes, its features, benefits, and how it enhances the networking capabilities of the platform.



1. What is CoreDNS?

CoreDNS is a cloud-native, open-source DNS server that provides service discovery and DNS-based load balancing for Kubernetes clusters. It is designed to be highly scalable, performant, and modular, making it an ideal choice for managing DNS within Kubernetes environments.

2. Features and Benefits of CoreDNS in Kubernetes:

a) Service Discovery: CoreDNS enables service discovery within Kubernetes by mapping service names to their corresponding IP addresses. It automatically updates DNS records as pods and services are created, scaled, or deleted, ensuring that the DNS resolution is always up to date.

b) DNS-Based Load Balancing: CoreDNS supports DNS-based load balancing, allowing traffic to be evenly distributed across multiple instances of a service. This load balancing mechanism ensures high availability and optimal performance for applications running in the cluster.

c) Plugin Architecture: CoreDNS follows a modular plugin architecture, making it highly extensible and customizable. It supports various plugins that can be added or removed based on specific requirements, such as caching, logging, metrics, and integration with external DNS providers.

d) Integration with Kubernetes: CoreDNS seamlessly integrates with Kubernetes, leveraging its APIs to retrieve information about pods, services, and endpoints. It can dynamically generate DNS records based on the current state of the cluster, eliminating the need for manual DNS configuration.

e) Fast DNS Resolution: CoreDNS is designed to be fast and efficient, ensuring low latency DNS resolution for optimal application performance. It can handle a high volume of DNS queries without compromising speed or reliability.

3. Configuring CoreDNS in Kubernetes:

a) Corefile: CoreDNS is configured using a file called the Corefile. The Corefile defines the DNS zones, plugins, and their configuration. It allows users to customize DNS behavior, add caching, logging, and other functionalities based on specific requirements.

b) Deployment: CoreDNS is typically deployed as a Kubernetes deployment or DaemonSet. It can be deployed as a standalone service or as part of a larger cluster deployment, depending on the specific networking requirements.

c) Service Configuration: In Kubernetes, the CoreDNS service is responsible for providing DNS resolution to pods and services within the cluster. By default, all pods in the cluster are configured to use the CoreDNS service as their DNS resolver.

4. Monitoring and Troubleshooting:

a) Logging and Metrics: CoreDNS provides logging capabilities to track DNS queries, errors, and other relevant information. Additionally, it supports metrics integration with monitoring systems like Prometheus, enabling visibility into DNS performance and health.

b) Troubleshooting: When troubleshooting DNS-related issues, it's important to check the CoreDNS logs for any errors or misconfigurations. DNS resolution can be validated using tools like nslookup or dig to ensure the expected IP addresses are returned.

**Livenss and readiness on Deployment:**

In a Kubernetes Deployment, liveness and readiness are two important properties that can be configured to ensure the health and availability of the deployed application. Let's explore what liveness and readiness mean in the context of a Deployment:

1. Liveness Probe:

The liveness probe in a Deployment is used to determine if an application instance within a pod is running correctly. It periodically checks the health of the application and takes action based on the probe's result. If the liveness probe fails, Kubernetes considers the application instance to be unhealthy and takes corrective measures, such as restarting the pod.

The liveness probe can be configured with different types of checks, such as an HTTP GET request, a TCP socket check, or an execution of a specific command within the container. The probe is considered successful if the defined check returns a healthy state (e.g., a successful HTTP response code, a successful TCP connection, or a specific exit status of the command).

2. Readiness Probe:

The readiness probe in a Deployment is used to determine if an application instance within a pod is ready to accept incoming network traffic. It ensures that the application instance is fully initialized and capable of serving requests before it starts receiving traffic from a Service or Ingress.

The readiness probe can also be configured with different types of checks, similar to the liveness probe. It periodically checks the readiness of the application, and if the probe fails, the application instance is considered not ready to serve traffic. Kubernetes will automatically exclude the pod from receiving traffic until the readiness probe reports a successful state.

By configuring both liveness and readiness probes in a Deployment, administrators can ensure that the application instances are running correctly and are ready to handle traffic. This helps in maintaining high availability, preventing traffic from being routed to unhealthy or not fully initialized instances, and providing a seamless user experience.

livenessProbe:

httpGet:

path: /health

port: 8080

initialDelaySeconds: 10

periodSeconds: 30

readinessProbe:

tcpSocket:

port: 8080

initialDelaySeconds: 5

periodSeconds: 10

In the above example, the Deployment defines a liveness probe that performs an HTTP GET request to the `/health` endpoint on port 8080 every 30 seconds. If the probe fails, Kubernetes will restart the pod after an initial delay of 10 seconds. Additionally, a readiness probe is configured to perform a TCP socket check on port 8080 every 10 seconds, with an initial delay of 5 seconds. If the readiness check fails, the pod will be excluded from receiving traffic until it reports a successful readiness state.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Feature | Liveness Probe | Readiness Probe |  |  |
| Purpose | Keeps a container running | Controls when a container is ready to receive traffic |  |  |
| Action taken on failure | Restarts the container | Removes the container from the service load balancer |  |  |
| Frequency | Can be run at any interval | Usually run less frequently than liveness probes |  |  |

**DeamonSet**

In Kubernetes, a DaemonSet is a type of workload that ensures that a specific pod runs on every node in the cluster. It is designed to provide a way to run a single instance of a pod on each node, ensuring that a particular task or service is running and available on all nodes.

Here are some key points to understand about DaemonSets:

1. Running on Every Node: When a DaemonSet is created, it automatically creates and schedules a pod on each node in the cluster. This ensures that the desired pod or workload is running on every available node.

2. One Pod per Node: Unlike other workload types, such as Deployments or ReplicaSets, which may have multiple replicas running on each node, a DaemonSet ensures that only one pod instance of the specified configuration runs on each node.

3. Pod Lifecycle: The DaemonSet manages the lifecycle of its pods. It automatically creates pods on new nodes that are added to the cluster and removes pods from nodes that are removed from the cluster. This ensures that the desired state is maintained across the cluster.

4. Use Cases: DaemonSets are commonly used for tasks that need to be performed on each node, such as monitoring agents, log collectors, or network proxies. They ensure that these services are available and running on every node, providing consistency and reliability across the cluster.

5. Node Affinity and Tolerations: DaemonSets support node affinity and tolerations, allowing you to define constraints for pod scheduling. This enables you to specify which nodes the DaemonSet should run on or exclude from running on based on node labels or other criteria.

6. Updates and Rollouts: When updates are required for a DaemonSet, you can use strategies like a rolling update to update pods on each node one by one, ensuring that the workload remains available throughout the update process.

7. Scaling: DaemonSets do not scale up or down like other workload types. If you want to add or remove nodes from a DaemonSet, you need to modify the underlying node pool or change the node selector or affinity rules.

Overall, DaemonSets provide a way to ensure that specific pods or workloads run on every node in a Kubernetes cluster. They are useful for running services or tasks that require node-level availability and are commonly used for monitoring, logging, or networking purposes.

**Resource Requests and Limits in Kubernetes**

Resource requests and limits are two important concepts in Kubernetes that allow you to control how much resources your pods can use.

Resource requests specify the minimum amount of resources that a pod needs to function properly. If a pod's request is not met, the pod may not be able to start or may not be able to perform its tasks.

Resource limits specify the maximum amount of resources that a pod can use. If a pod exceeds its limit, it will be evicted from the node. This can happen if the pod is using too much memory, CPU, or other resources.

It is important to set appropriate resource requests and limits for your pods. If you set too low of a request, the pod may not be able to start or may not be able to perform its tasks. If you set too high of a limit, the pod may use more resources than it needs, which can lead to performance problems or even node evictions.

**Comparison of Resource Requests and Limits**

The main difference between resource requests and resource limits is that a request is the amount of resources a pod needs, while a limit is the maximum amount of resources a pod can use.

|  |  |  |
| --- | --- | --- |
| Feature | Resource Requests | Resource Limits |
| Definition | The minimum amount of resources a pod needs to function properly. | The maximum amount of resources a pod can use. |
| Impact | If a pod's request is not met, the pod may not be able to start or may not be able to perform its tasks. | If a pod exceeds its limit, it will be evicted from the node. |
| Relationship | A request must always be less than or equal to a limit. | A limit can be set to a specific value or to "", which means that there is no limit. |
| Best practices | Set requests to the minimum amount of resources that the pod needs to function properly. | Set limits to the maximum amount of resources that the pod should use. |

**Behavior of Pods When They Get Limits**

When a pod gets a limit, it means that it is not allowed to use more than a certain amount of resources. This can be due to a number of factors, such as the node's capacity or the pod's own configuration.

If a pod exceeds its resource limits, it will be evicted from the node. This can happen if the pod is using too much memory, CPU, or other resources.

There are a few things that can happen when a pod gets a limit:

* The pod may be evicted from the node.
* The pod may be throttled, which means that it will be prevented from using more resources.
* The pod may continue to run, but it may experience performance problems.

The specific behavior of a pod when it gets a limit will depend on the Kubernetes scheduler and the pod's configuration.

**Types of Resource Limits in Kubernetes**

Kubernetes has different types of resource limits that can be applied to pods:

1. CPU limits: These limits specify the maximum amount of CPU that a pod can use.
2. Memory limits: These limits specify the maximum amount of memory that a pod can use.
3. Storage limits: These limits specify the maximum amount of storage that a pod can use.
4. GPU limits: These limits specify the maximum number of GPUs that a pod can use.
5. Network limits: These limits specify the maximum amount of network bandwidth that a pod can use.

Resource limits can be set on a per-pod basis or on a per-namespace basis. When set on a per-pod basis, the limits apply to all containers in the pod. When set on a per-namespace basis, the limits apply to all pods in the namespace.

Resource limits are enforced by the Kubernetes scheduler. When the scheduler is scheduling a pod, it will only consider nodes that have enough resources to satisfy the pod's resource limits.

**When to Use Resource Limits**

Resource limits should be used when you want to control the amount of resources that pods can use. This is important for pods that are known to be resource-intensive, such as database pods or media streaming pods.

Resource limits can also be used to prevent pods from using all of the resources on a node. This is important for pods that are not known to be resource-intensive, but that you want to make sure do not have a negative impact on other pods on the node.

**How to Set Resource Limits**

To set resource limits, you can use the limits field in the pod's spec. The limits field specifies the maximum amount of resources that the pod can use.

For example, the following pod spec sets a CPU limit of 1000 millicores and a memory limit of 100 megabytes:

apiVersion: v1

kind: Pod

metadata:

name: my-pod

spec:

containers:

- name: my-container

image: nginx

resources:

limits:

cpu: 1000m

memory: 100Mi

In this example, the pod is allowed to use up to 1000 millicores of CPU and up to 100 megabytes of memory.

It is important to set appropriate resource limits for your pods. If you set too low of a limit, the pod may not be able to start or may not be able to perform its tasks. If you set too high of a limit, the pod may use more resources than it needs, which can lead to performance problems or even node evictions.

**Afiinity**

In Kubernetes, affinity is a feature that allows you to control where pods are scheduled. There are two types of affinity:

* Node affinity allows you to constrain which nodes a pod can be scheduled on based on the labels on those nodes.
* Pod affinity allows you to constrain which nodes a pod can be scheduled on based on the labels on other pods.

Node affinity is similar to the nodeSelector field, but it is more expressive and allows you to specify soft rules. Pod affinity allows you to constrain pods against labels on other pods.

For example, you could use node affinity to ensure that a pod is only scheduled on nodes that have a certain amount of memory or CPU. You could also use pod affinity to ensure that two pods are always scheduled on the same node or that they are never scheduled on the same node.

Affinity can be used to improve the performance, reliability, or security of your Kubernetes applications. For example, you could use affinity to ensure that your database pods are always scheduled on nodes that have high availability or that your web pods are never scheduled on nodes that are in the same region as your database pods.

Here is an example of how to use node affinity in a Kubernetes manifest:

apiVersion: v1

kind: Pod

metadata:

name: my-pod

spec:

affinity:

nodeAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

nodeSelectorTerms:

- matchExpressions:

- key: disktype

operator: In

values:

- ssd

This manifest specifies that the pod must be scheduled on a node that has the disktype=ssd label.

Here is an example of how to use pod affinity in a Kubernetes manifest:

apiVersion: v1

kind: Pod

metadata:

name: my-pod

spec:

affinity:

podAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

- podAffinityTerm:

labelSelector:

matchExpressions:

- key: security

operator: In

values:

- S1

This manifest specifies that the pod must be scheduled on a node that is in the same zone as one or more pods that have the security=S1 label.

**Taint on K8s**

In Kubernetes, a taint is a way to mark a node so that it can only be scheduled to pods that have a matching toleration. Taints are used to control where pods are scheduled in a Kubernetes cluster.

A taint is defined by a key and an effect. The key is a string that identifies the taint. The effect can be one of three values:

1. NoSchedule: This prevents pods from being scheduled on the node.
2. PreferNoSchedule: This tells the scheduler to prefer not to schedule pods on the node, but it will still do so if there are no other nodes available.
3. NoExecute: This evicts any pods that are already running on the node.

Tolerations are defined by a key and an effect. The key must match the key of a taint. The effect can be one of three values:

1. Equal: This means that the pod tolerates the taint.
2. Exists: This means that the pod tolerates any taint with the specified key.
3. None: This means that the pod does not tolerate any taints.

When a pod is scheduled, the scheduler will first check if the pod has any tolerations that match the taints of the node. If the pod has a toleration that matches the taint, the pod will be scheduled on the node. If the pod does not have a toleration that matches the taint, the pod will not be scheduled on the node.

**etcd on k8s**

Etcd is a distributed reliable key-value store which is simple, fast and secure. It acts like a backend service discovery and database, runs on different servers in Kubernetes clusters at the same time to monitor changes in clusters and to store state/configuration data that should to be accessed by a Kubernetes master or clusters. Additionally, etcd allows Kubernetes master to support discovery service so that deployed application can declare their availability for inclusion in service.

The API server component in Kubernetes master nodes communicates with etcd the components spread across different clusters. Etcd is also useful to set up the desired state for the system.

By means of key-value store for Kubernetes etcd, it stores all configurations for Kubernetes clusters. It is different than traditional database which stores data in tabular form. Etcd creates a database page for each record which do not hampers other records while updating one. For example, this might happen that few records may require additional columns, but those not required by other records in the same database. This creates redundancy within database. Etcd adds and manages all records in reliable way for Kubernetes.

etcd is used by Kubernetes to store a variety of data, including:

1. Configuration information: This includes information such as the cluster's name, the list of nodes in the cluster, and the Kubernetes version.
2. API server endpoints: This includes the addresses and ports of the API servers in the cluster.
3. Cluster membership information: This includes information about which nodes are members of the cluster.
4. Other data: etcd can also be used to store other data, such as secrets or configuration data for applications.
5. etcd is a critical component of Kubernetes, and it is important to understand how it works and how to use it.

**Deployment Methods for etcd in Kubernetes Clusters**

Etcd is implementation is architected in such a way to enable high availability in Kubernetes. Etcd can be deployed as pods in master nodes

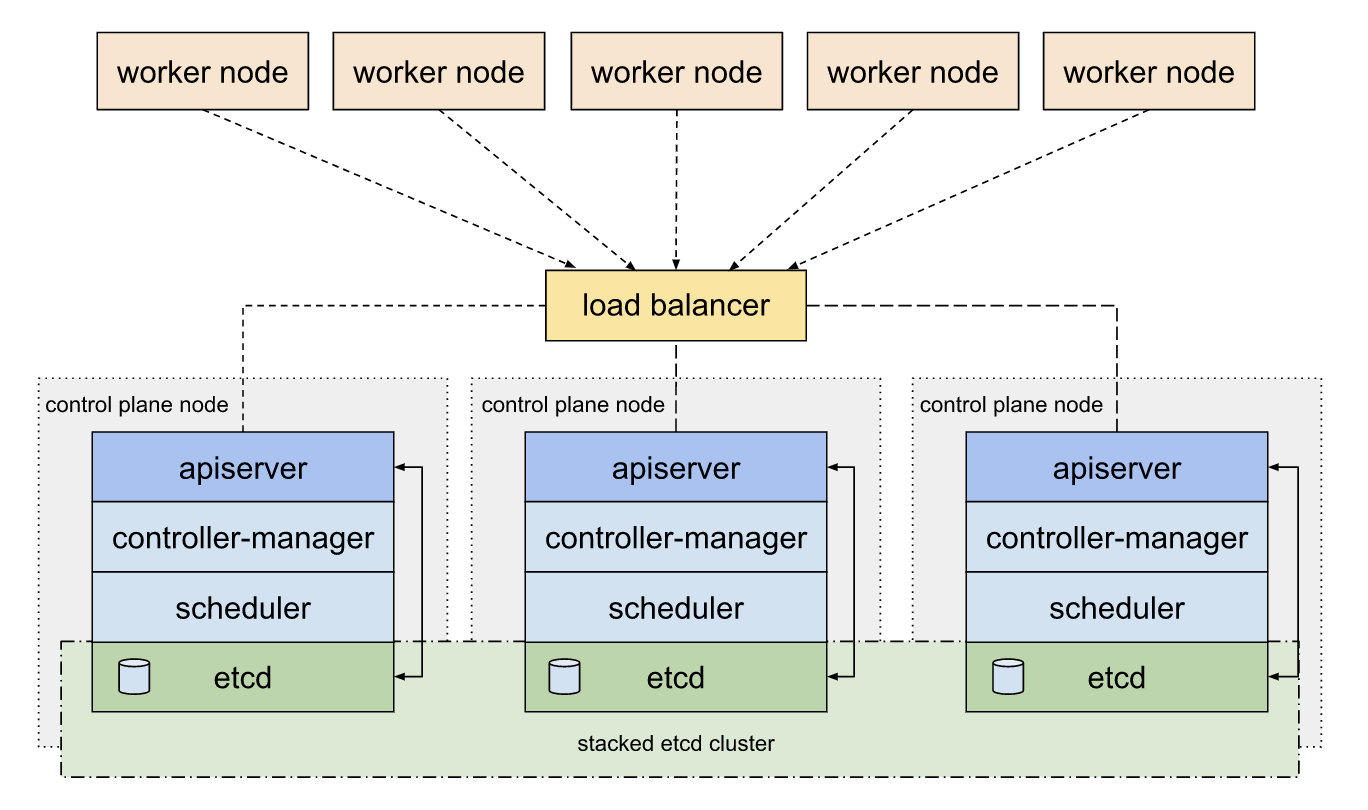


Figure 1 etcd in the same cluster

It can also be deployed externally to enable resiliency and security

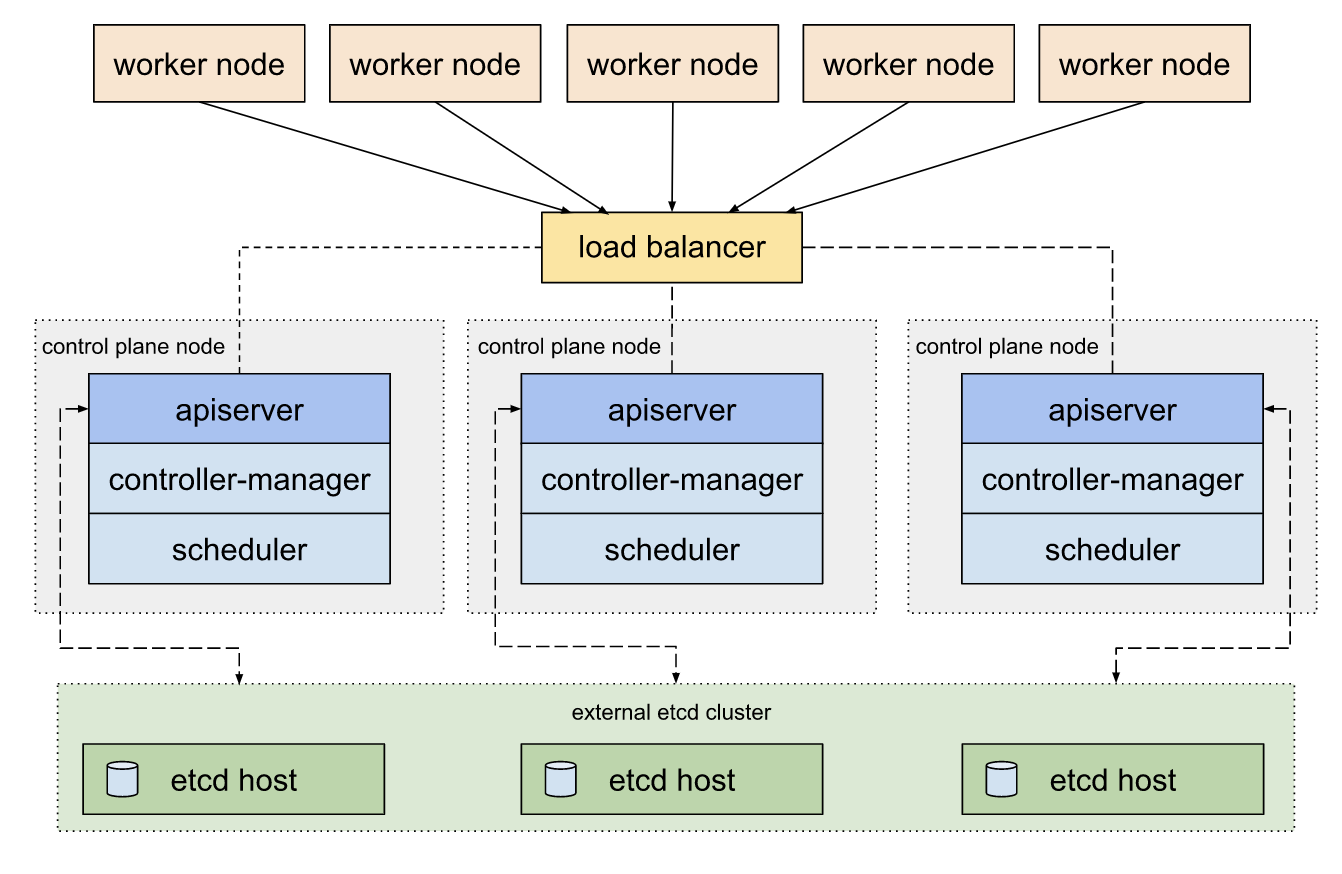


Figure 2 etcd deployed externally

|  |  |  |
| --- | --- | --- |
| Feature | etcd | External etcd |
| Definition | etcd is a distributed key-value store used as the Kubernetes cluster's central store for storing data such as configuration information, API server endpoints, and cluster membership information. | External etcd is a deployment of etcd that is not managed by Kubernetes. It can be used to store data for Kubernetes, but it is not required. |
| Deployment | etcd is typically deployed as a cluster of three or more nodes. | External etcd can be deployed as a cluster of any size, or it can be deployed as a single node. |
| Advantages | Simplicity: etcd is a simple and easy-to-use key-value store. | Flexibility: External etcd can be deployed in any way that you want. |
| Disadvantages | Single point of failure: If one node in an etcd cluster fails, the entire cluster can become unavailable. | Complexity: Managing an external etcd cluster can be more complex than managing an etcd cluster that is managed by Kubernetes. |